TAMPERE UNIVERSITY OF TECHNOLOGY

ALEKSI KALLIO

AUTOMATED WEB STORE PRODUCT SCRAPING USING NODE.JS

Master of Science Thesis

Examiner: Prof. Tommi Mikkonen
Examiner and topic approved by the Council of the Faculty of Computing and Electrical Engineering on 8th of April 2015

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
Aleksi Kallio : Automated web store product scraping using Node.js
Master of Science Thesis, 48 pages
June 2015
Major: Software Engineering
Examiner: Prof. Tommi Mikkonen
Keywords: Web, eCommerce, HTML, JavaScript, Node.js, Web crawler, NoSQL

Different fields of electronic commerce have grown substantially in the last decade. This is mainly due to increased accessibility of internet and the improvements in other network technologies. Also, the abundance of mobile devices has made the electronic commerce easily accessible for everyone, from anywhere, at any time. The biggest form of electronic commerce is online shopping, which is a huge and steadily growing world wide business.

The growth of online shopping brings new possibilities for market research and behavioural research. The data from online shopping could, for example, be used to study price changes and commodity consumption across the globe. To study these globe wide phenomena, large quantities of online shopping data is needed. The product catalogues of the online stores are especially well suited for multitude of different researches. To gain large quantities of information from these product catalogues, it should be possible to acquire product catalogues from multiple stores automatically and reliable, over a significant timespan and for multiple consecutive times.

In this thesis a web store product scraper software, capable of collecting product catalogue information from several web stores, was implemented. The software was implemented using JavaScript programming language, NodeJS framework, MongoDB NoSQL database and multiple well proven software development architectures. The web store product scraper was configured and tested with several different settings on three different sized web stores. The results were promising. From each store a significant amount of products were scraped. The amounts were also in line with the sizes of the stores. The stores were scraped concurrently and simultaneously without supervision and with low impact on system resources.

Collecting product information from online stores is possible and well proven, even though collecting information from large web stores takes time. The information can be scraped concurrently and simultaneously from multiple web stores. Future work should be more concentrated on building a framework around the web store product scrapers than to optimise the system resource consumption. The framework should simplify the configuration and monitoring of multiple simultaneous web store product scrapers.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
ALEKSI KALLIO: Automaattinen tuotteiden informaation kaavinta nettikaupasta käyttäen Node.js -ohjelmistokehystä
Diplomityö, 48 sivua
Kesäkuu 2015
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Tommi Mikkonen
Avainsanat: Web, eCommerce, HTML, JavaScript, Node.js, Web crawler, NoSQL

Elektronisen kaupankäynnin eri alalajit ovat kasvaneet nopeasti. Suurimpana syynä tähän on ollut internetin saavutettavuuden parantuminen sekä muiden verkkoteknologioiden nopea kehittyminen. Elektronisen kaupankäynnin suurin alatyyppi on verkkokauppa, joka kasvaa maailmanlaajuisesti valtavasti joka vuosi. Verkkokaupan suuri kasvu synnyttää uusia mahdollisuuksia myös markkina- ja käyttäytymistutkimuksille. Verkkokaupasta syntyvää tietoa voitaisiin esimerkiksi käyttää maailmanlaajuisten trendien ja hinnan kehitysten tutkimukseen.

Näiden maailmanlaajuisten ilmiöiden tutkimukseen tarvitaan suuri määrä verkkokauppaan liittyvää informaatiota. Verkkokauppojen tuotetiedot ovat erityisen hyviä tiedonlähteitä monille eri tutkimuksille. Jotta verkkokauppainformaatiota olisi helppo hankkia, sitä pitäisi pystyä keräämään monesta eri lähteestä automaattisesti ja varmasti. Informaatioita pitäisi myöskin pystyä hankkimaan toistuvasti pitkältä aikaväliltä.

Tässä työssä kehitettiin verkkokauppojen tuotetiedon kaavintaohjelma, joka pystyy keräämään tuotetietoa lukuisista verkkokaupoista. Ohjelma toteutettiin JavaScript -ohjelmointikielellä käyttäen Node.js -ohjelmistokehystä, MongoDB NoSQL -tietokantaa sekä lukuisia hyväksi todettuja ohjelmistoarkkitehtuureja ja malleja. Verkkokauppojen tuotetiedon kaavintaohjelma määritettiin ja testattiin monilla eri asetuksilla käyttäen kolmea eri kokoista verkkokauppaa. Tulokset olivat lupaavia. Jokaisesta verkkokaupasta saatiin kerättyä merkittävä määrä tuotetietoa, ja kerätyn tuotetiedon määrä oli myös linjassa verkkokauppojen koon kanssa. Kehitetyn ohjelman avulla verkkokauppoja oli mahdollista kaapia rinnakkain ja samanaikasesti ilman ulkopuolista valvontaa. Kaavinta myös kulutti vähän järjestelmän resursseja.

Verkkokauppojen tuotetiedon järjestelmällinen kerääminen on mahdollista ja todistettua. Vaikka suurien verkkokauppojen kaavintaan kuluu paljon aikaa, niin monia verkkokauppoja voidaan kaapia rinnakkain ja samanaikaisesti. Tulevaisuuden työ pitäisi keskittää järjestelmän resurssien kulutuksen optimoinnin sijasta kaapijan ympärillä olevan ohjelmistokehyksen kehittämiseen. Ohjelmistokehys helpottaisi kaapijan verkkokauppa kohtaisten asetusten määrittelyä sekä lukuisten samanaikaisten kaapijoiden valvontaa.

PREFACE

This thesis was carried out while working for a customer project in Vincit Oy in Tampere, Finland.

I would like to thank all my colleagues at Vincit and the customer for introducing me to the subject and giving help when needed. Special thanks to Olli Salli who was a big help on behalf of Vincit and Professor Tommi Mikkonen from Tampere University of Technology, who was the examiner of this thesis.

Finally, I want to thank Tiina, who listened to my problems and helped me to put my thoughts on paper. I want to thank also all my friends, who gave me the balance between work and leisure.

Tampere May 17, 2015

Aleksi Kallio

TABLE OF CONTENTS

TERMS AND DEFINITIONS

CSS	Cascading Style Sheet
DOM	Document Object Model
eCommerce	Electronic commerce
EDA	Event-driven architecture
HTML	HyperText Markup Language
I/O	Input-Output
JSON	JavaScript Object Notation
Node.js	A platform to run JavaScript without a browser.
NoSQL	Not only SQL
RDBMS	Relational Database Managing Systems
SOA	Service oriented architecture
SQL	Structured Query Language
W3C	World Wide Web Consortium
XML	Extensible Markup Language

1. INTRODUCTION

Different fields of electronic commerce have grown substantially in the last decade due the increased accessibility of internet and other network technologies. Electronic commerce, also known as eCommerce, can be generalized as a type of commerce industry that takes place over electronic systems, usually over the internet. Online shopping is the biggest form of eCommerce. ECommerce also encapsulates many other commerce technologies such as mobile shopping, internet marketing and electronic data interchange. Online shopping allows consumers to buy physical products or services ranging from clothes and electronics to travel tickets and hotel nights using a web browser. Online shopping is huge and steadily growing world wide business. Only in United States online retail sales accounted for almost 9% of the $3.2 trillion total retail market in 2013. Online retail market is expected to grow nearly 10% annually through 2018. [1]

The huge growth of online shopping brings new possibilities for market research and behavioural research. The online shopping data could be used to study price changes and commodity consumption across the globe, and to identify consumption patterns and rising trends. These globe wide researches concentrate more on the large changes and unifying characteristics of the data, rather than small individual changes. To yield meaningful research results, large quantities of online shopping data is needed. Especially the product catalogues of the online stores can be used for a multitude of different researches. To research these possibilities, a lot of product catalogue data is needed from multiple sources and across meaningful timespan.

In this thesis we introduce and implement a concept to collect product catalogue data from a vast amount of web stores concurrently and repeatedly. The scope of the data acquisition in this thesis is restricted to online shopping websites that offer physical products, e.g. clothes and electronics, but it could easily be developed to also contain other branches of online shopping.

This thesis consists of seven chapters. Chapter 2 dives in to the problem of acquiring the product catalogue information from multitude of different web stores. It also introduce a software concept to solve the problem. Chapter 3 introduces the language and framework in which the software will be implemented in. Chapter 4 is about the software development architectures and patterns by which the software will be implemented in. Chapter 5 discusses about the implementation of the software. Chapter 6 is reserved for testing the software with a set of real web stores and also some points for future work is given. Chapter 7 concludes the thesis with some final remarks.

2. WEB STORES

There are a lot of online stores in the web. These web stores vary in their size from stores with tens of products to huge stores with tens of thousands of products. These stores vary also in the amount of customers and the infrastructure. Even though each store is built for a different purpose, the stores have similarities in their framework and composition.

2.1 Web Store Platforms

Web stores are usually built on a specific eCommerce system. This system is a collection of different software systems that encapsulate all the necessary components and functions of the web store. An eCommerce system is usually modelled with three-tier architecture. It usually includes a database for the products, software to handle the business logic, and web server to serve the web pages for the consumers. These web pages are also known as storefronts. Figure 2.1 illustrates an outline of a possible eCommerce system.

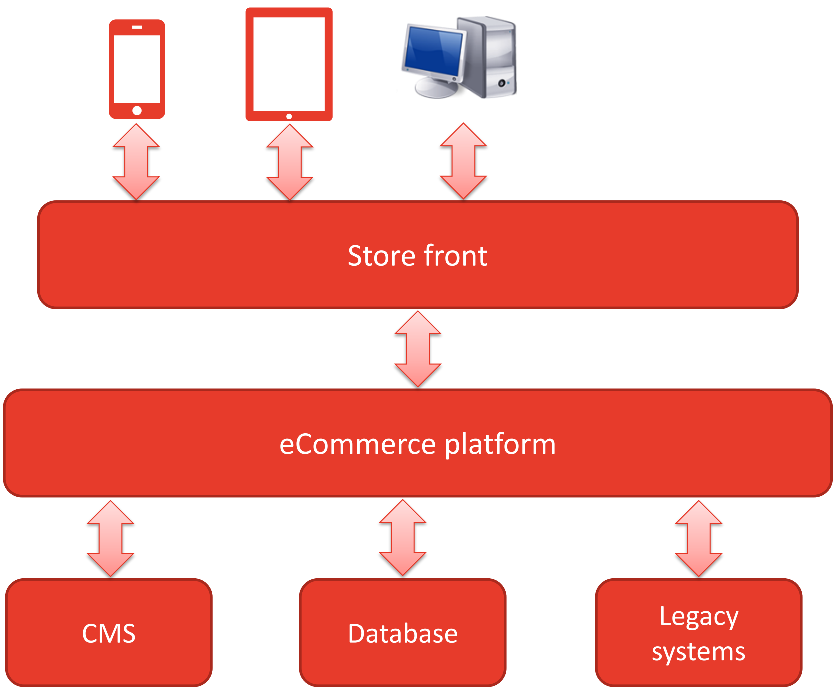

Figure 2.1: eCommerce platform.

At the bottom of the eCommerce system exists multiple individual components for multiple specific tasks. Usually there are at least a content management system (CMS) for managing the web pages and other content, and a database for managing the product catalogue and storing customer information. In addition to these, there can be other legacy systems specialized to different aspects of commerce, e.g. marketing or product stock management systems. On top of those is eCommerce platform software that integrates individual legacy systems. It handles the communication between individual components and also between the customer and the legacy systems. At the top of the eCommerce system is the storefront, which is the only part that is visible to the customer. It consists of web server, which takes care of serving the web pages to the shopper. The top of Figure 2.1 represents different client applications and devices connecting to storefront. Even though customers can use a multitude of different devices and applications to access the web store, all these systems communicate through the storefront and web server. [2]

The underlying eCommerce platform does not enforce a certain look or functionality to the storefront. Usually platforms offer a couple of ready-made themes or templates, from which the eCommerce platform user can customize their storefront. In reality, even if two storefronts look completely different the underlying structure of the web page can be similar, it is only styled differently. This is usually the case with small web stores, which do not have neither the resources nor the skills to make unique storefronts themselves. On the other hand, storefronts of big web store companies vary much more in their looks and functionality. Usually those are custom made and might have no resemblance to other storefronts using the same eCommerce platform.

2.2 Information Sources

There are three popular data formats from which it is possible to obtain product catalogue information for a web store. Usually product catalogues are in HyperText Markup Language (HTML) or Extensible Markup Language (XML) format. In some stores it is also possible to acquire the product data in JavaScript Object Notation (JSON) format.

HTML files are the storefront files that the web server of the eCommerce platform serves to the shoppers. These can be easily obtained from the web server by requesting them through HTTP protocol e.g. using a browser. XML formatted product data files are harder to obtain as they are not usually publicly available and not all web stores support them. Larger multi-national web stores usually have so called affiliate program. This affiliate program is originally meant to promote the products of the web store by advertising them on different web sites such as blogs or news pages. When someone clicks these advertisements, the user is forwarded to the web store, and the original advertiser, e.g., the blog writer, gets a small commission. This commission can come from simply forwarding other users to the web store or it can come from the actual purchase of a product. As these affiliate programs usually have contracts, which dictate that to use the product data, the user must also show advertisements. Because of that, using the product data from an XML file without showing any advertisements may be an issue. JSON formated product data can usually be acquired in the same way as XML formated data, and it is used for the same purpose. The available JSON data, however, is even more scarce than XML fromated data. The HTML formated storefronts are available to everyone, so those are the main focus of this thesis. [3; 4]

2.2.1 Extensible Markup Language

Extensible Markup Language (XML) is a markup language that is mainly used to store and to move data. XML defines a set of rules for encoding the data. This format is readable by both humans and machines. XML standard has two versions 1.0 and 1.1. Version 1.0 was initially defined in 1998, and it is currently in its fifth edition. Versopn 1.1 was published in 2004, and it is currently in its second edition which was published in 2006. XML versions 1.0 and 1.1 are similar to each other. The main differences are that version 1.1 allows the use of scripts and characters that are absent from Unicode version 3.2. The small difference between XML versions 1.0 and 1.1 has caused the 1.1 to have few implementations. It is recommended to use the 1.0 version unless there is a need for the special features of version 1.1. There have been some plans for XML 2.0 but at the moment there is no standard for it. [5]

An XML document consists of markup and content. Markup is the characters that define and describe the content of the document. Markup consists of tags and elements. A tag consists of angle brackets (<>) and a identifier between them. There are three different tags: a start tag, end-tag, and empty-element tag. Listing 2.1 presents a simple "Hello world" data structure with XML.

```xml
<Greeting name="Joe"> <!-- A Greeting tag with an attribute 'name',
    which has value 'Joe' -->
  <Message>Hello, world.</Message>
</Greeting> <!-- End of the Greeting tag -->
```

Listing 2.1: A Hello world example with XML

Tag identifiers can have any unicode characters in them. Elements are the main components of a XML document. Elements start with a start-tag and end with an end-tag or consist only from empty-element tag. Elements can have content between the start and end tag. The content consists of unicode characters which can then form other elements. These nested elements are called *child elements*. Nested elements can be used to construct very complex document structures. Tags can have attributes, which are simple key value pairs. Each attribute can have a single value, and the same attribute can only appear once on each element. Attributes can be used to include metadata to elements content. [6]

2.2.2 HyperText Markup Language

HyperText Markup Language (HTML) is the main language used in web pages. HTML defines the markup and content of a web page. It does not influence to how a web page looks or functions. The first HTML standard, 2.0, was released in 1995. In 1997, version 3.2 was released which was the first version by World Wide Web Consortium (W3C). W3C released HTML 4.0 in 1997 and it got a minor upgrade in 1999 to 4.01. Today, 4.01 is still the most recent standard for HTML. W3C is currently developing HTML version 5 and has released a candidate recommendation of it in December 2012. Although HTML 5 is not yet a standard, many browsers already support some of its new features. [7; 8]

HTML is a markup language similar to XML. As XML, also HTML consists of tags and elements, which can be combined to construct complex documents. In HTML, however, the tag names are strictly named and standardized by W3C. HTML also specifies certain named attributes for the tags, e.g. `id` and `class`, which can be used to distinguish or group different elements to logical groups. Named tags of HTML add an another meaning to the markup of the document as they also define a purpose for the element. For example, <h2> defines a second-level heading. HTML does not denote any specific rendering rules for the document. Listing 2.2 illustrates a simple hello world with HTML. [8]

```
<!DOCTYPE html>
<html>
  <head>
    <title>This is a title of the document</title>
  </head>
  <body>
    <h2 id="greeting" class="message">Hello world!</h2>
  </body>
</html>
```

Listing 2.2: A hello world example with HTML.

2.2.3 JavaScript Object Notation

JavaScript Object Notation (JSON) is a simple, text-based data format that is based on the `object` data type of the JavaScript programming language standard.

JSON data elements consists of four primitive types: strings, numbers, booleans, or nulls. In addition to these JSON elements can also consist of two different structure type: objects, and arrays. All these structures can be found in almost all modern programmin languages in one form or another. This makes JSON interchangeable with other programming languages and easy for humans and machines to read and write.

The basis of a JSON data format is an object that consists of unordered collection of name-value pairs. The name in the pair is a string and the value can be any other JSON type, even another object. Every object starts with a left brace ({) and ends with a right brace (}). Each name of the object is inside quotation marks ("") and is followed by a colon (:) and a value. The name-value pairs are separated by comma (,). Unlike in JavaScript language, in JSON the formating of name-value pairs is strict. Missing quotation marks or unnecessary comma in the end of a list will produce an error. Listing 2.3 presents a simple "Hello world" datastructure in JSON. [9; 10]

```
{
  "Greeting": {
    "Message": "Hello World!"
  },
  "Receiver": "John"
}
```

Listing 2.3: A hello world example with JSON.

2.3 Acquiring Product Information

A notable difference between HTML, XML and JSON formatted product catalogue data is that the XML and JSON formatted product data usually holds the entire product catalogue of the web store in a single file. The HTML document only describes a single web page and thus the information of a single product. To acquire the entire product catalogue as HTML documents, one document must be downloaded for each product.

In this thesis we concentrate on collecting product information through HTML files, as those are easier to acquire. Collecting product information is a two step process: first the HTML file containing the product is downloaded from the web server of the web store. Then, the HTML code of the file is processed to extract the necessary information. The problem is, how to methodologically download and process every HTML file from a web store.

Web crawler (or web spider) is a software that can be used to reliably download a set of web pages. The crawler is initiated with a page and it will continue from there, downloading every web page linked to the original page. The crawler will continue until there are no new pages left.

After the HTML file of a product is downloaded, it has to be processed to filter out any non-relevant information. This can be done by first analysing the HTML code and extracting the important HTML elements. The HTML elements are then analysed for relevant information. Finally the information is processed to a suitable form and stored to a database.

3. NODE.JS

JavaScript is the language of the web and it is usually run by the browser. Node.js (later Node) is a platform that allows running the JavaScript code without a browser. Node is built on the same V8 JavaScript virtual machine as Google Chrome. With V8 JavaScript is compiled into native machine code, instead of interpreting it as bytecode. This compiled machine code is also dynamically optimized during the runtime. This boosts the performance of Node over browsers. Node uses an event-driven non-blocking input-output (I/O) model that makes it lightweight and very efficient [11]. In this chapter we take a look into Node. First we discuss about JavaScript and its benefits and fall-backs. Then we will see what makes Node so efficient and good choice for dataintensive applications. [12]

3.1 Javascript Basics

JavaScript is one of the world's most used programming languages as it is in use on almost every modern web page. JavaScript handles all the functionality taking place on a web page and thus it is becoming almost unavoidable to do any programming in the web without coming across JavaScript.

When JavaScript first came out in mid 90's it was only used for little visual enhancements on websites. By 2005, when Ajax (Asynchronous JavaScript And XML) revolution came, JavaScript evolved from being a "toy" language to something people wrote real code with. Ajax is a collection of techniques that allow asynchronous data interchange between the browser and the server. Google Maps and Gmail were some of the first applications written, that made use of Ajax. In 2008, Google Chrome was released to compete with other browsers. This lead to JavaScript performance taking a big leap due to the V8 virtual machine. Since then JavaScript performance has improved on a incredibly fast rate due to browser competition. One example of the big performance improvement of JavaScript is JSLinux. JSLinux is a PC emulator running on JavaScript that can load Linux kernel, interact with console and even compile C programs, all in a browser. [12]

JavaScript is a scripting language, which can be used to implement multiple programming language paradigms: scripting, object-oriented, imperative and functional. Syntactically JavaScript resembles C, C++ and Java, as it has similar syntax for `if` and `loop` statements. JavaScript statements also end with a semicolon (;).

JavaScript interpreters add missing semicolons automatically, but not always where the programmer intended. Because of that it is good practice to always end statements with a semicolon. JavaScript is dynamically typed language and new variables are defined with var keyword. JavaScript has the following different types: Number, String, Boolean, Array, and Object. In JavaScript, arrays and functions are descendants of the Object type. This makes also functions first class citizens and allows them to be passed and returned as function parameters. In JavaScript, there is no built-in I/O functionality. Instead the runtime environment, e.g. browser, provides the I/O functionality. [13]

3.2 Node Fundamentals

As already mentioned, Node is a platform that allows running JavaScript code without the browser. Node works similarly to other scripting language interpreters, e.g. Python and Perl. It can be used as Read-Eval-Print-Loop (REPL) straight from the console or it can be used to launch JavaScript files. Node supports newest ECMAScript specification and common browser functions e.g. console. [12]

In order to make JavaScript function in a browser, it is necessary to add the JavaScript files to the HTML through <script></script> tags. In Node, there are no HTML files and JavaScript language does not define any ways to include other JavaScript files. Node overcomes this through require function, which enables including other modules. In many other languages, including of other files may pollute the global namespace with unwanted variables or even overwriting others. Usually this is handled with different namespaces. In JavaScript, however, there are no namespaces. Node handles this by allowing developers to assign functions to be included as properties of a variable exports. If only one function is to be included it can be assigned to module.export variable. When require function is called, the exports object gets returned. This object can then be assigned to arbitrary named variable. Listing 3.1 defines a Node module that has two functions: area and circumference. Listing 3.2 requires the circle.js file and get the two functions as the attributes of the circle variable.

```
var PI = Math.PI;

exports.area = function(rad) {
  return 2*PI*Math.pow(rad,2);
}
exports.circumference = function(rad) {
  return 2*PI*rad;
}
```

Listing 3.1: Defining a Node module circle.js

```
var circle = require('circle.js');
var area = circle.area(2)

console.log(area); //Would output 8*PI
```

Listing 3.2: Requiring circle module

Another special aspect of Node is its command flow, which is asynchronous and event driven. Unlike in today's common concurrency model, where server applications employ multiple OS threads, Node runs in only one thread. It is possible to run Node in multiple threads, but it is often unnecessary. Node accomplishes this by employing a non-blocking event-loop. In other common server side programming languages I/O tasks almost always block code execution, but in Node this is not true. Because of that, programmers do not need to worry about deadlocking the system. [12]

For example, in synchronous command flow the following database query stops the whole code from executing until it is complete:

```
$data = mysql_query('SELECT * FROM myTable');
print_r($data);
```

Listing 3.3: Execution blocking database query with PHP

This query halts the whole process for the duration of the query. If there are other tasks to handle, the server would typically use a multi-threaded approach to allocate one thread for each task. In bigger applications, managing and allocating different threads can become very difficult. Also a large number of threads can spend a lot of system resources to perform context switches across different requests. In asynchronous command flow, a similar database query to Listing 3.3 would be written as the following:

```
mysql.query('SELECT * FROM mytable', function(err, result) {
  if (err) throw err;
  console.log(result);
});
```

Listing 3.4: Non execution blocking database query with JavaScript

In Listing 3.4, the execution of the code continues after the request to the database is done. When the database query returns, a callback function is executed with the query data. After the callback function is executed, the code continues execution where it was. This allows Node to handle multiple tasks in a single thread. In Node, almost all I/O operations occur outside the main event loop. This allows the server to stay efficient and ready to handle new requests. It also makes the server quite simple and straightforward to implement.

The event-loop behaviour of Node works similarly to JavaScript event-loop in browsers. The event-loop of Node is depicted in Figure 3.1. In step 1, the same database query is made as in Listing 3.4. Then in step 2, disk is read for some information and it is processed. In step 3, another user connects to the system and gets a respond. And after this in step 4, our database query comes back and a callback function is executed. Because of the non-blocking event-loop of the Node, this all occurs in one thread. In other common sever side programming languages, all this would have needed at least two threads. [12]

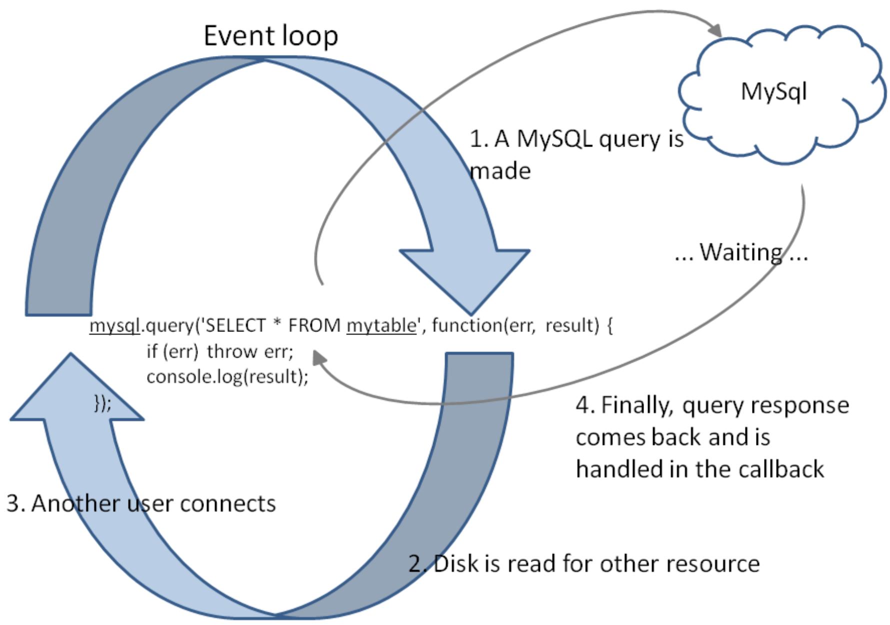

Figure 3.1: Node event loop

3.3 Asynchronous Techniques in Node

Because of the asynchronous event-loop, asynchronous functions and asynchronous coding style is very common in Node. In asynchronous command flow, the order in which functions are called is not predefined and it can vary between executions. This may raise some problems and needs some time to get used to. Node programming can be thought as similar to the browser JavaScript, events occur that trigger response logic. In Node, there are two popular models for handling event response logic: callbacks and event listeners.

3.3.1 Callbacks in Node

Callbacks are functions that are passed as arguments to asynchronous functions. Callbacks define the response logic for one-off responses. Callbacks can be used for, e.g. displaying results of a database query. Usually callbacks are used as anonymous single use functions, but of course they can also be named and reused. Listing 3.5 demonstrates the use of anonymous callbacks. First a HTTP server, which listens to port 8000, is created. A request to the root fires a query to the database. The database query calls its callback function to write the result to disc, which will then log ’done’ to the console.

```
http.createServer(function(req, res) {
  if (req.url === ’/’) {
    mysql.query(’SELECT * FROM mytable’, function(err, result) {
      if (err){
        throw err;
      } else {
        fs.writefile(’outputfile.txt’, result, function(err) {
          if (err) {
            throw err;
          } else {
            console.log("done");
          }
        });
      }
    });
  }
}).listen(8000, "127.0.0.1");
```

Listing 3.5: Example of JavaScript callbacks

Listing 3.5 has three levels of callbacks which is tolerable, but sometimes there can be even more levels. Multiple nested callbacks can make the code hard to read, maintain and test. One way to make the code more readable and maintainable is to use named functions for each callback. This is especially useful, if several of the callbacks are similar. Code nesting can also be decreased by reducing `if/else` blocks by using common Node idiom: returning early from a function. This means that if an error occurs, instead of writing the `else` statement, the code would return in the end of `if` block.

A notable thing in Listing 3.5 is the parameters of the callbacks. Most Node built-in modules use callbacks with two arguments. The first argument is an error, if one has occurred, and the second argument is the result of the query. This convention is also widely used by third-party modules. [12]

3.3.2 Events in Node

Events are fired up by event emitters and caught by event listeners. Event emitters also have the ability to listen to other events. Event listeners are a association of a callback function to an certain event. The callback function gets triggered every time the event occurs. Events are useful as they can have multiple different listeners. The emitters and the listeners do not need to know about each other. Many Node API components are implemented as event emitters, e.g. different servers and streams. Event emitters are also easy to make by inheriting them from the event emitter base class. Listing 3.6 implements a simple echo server. Whenever a client connects to it a socket is created. A socket is an event emitter that can have listeners added to it. In this case a listener is added for the *'data'* event. Every time the socket receives new data it will echoe it back to the client.

```
var server = net.createServer( function(socket) {
  socket.on('data', function(data) {
    socket.write(data);
  });
});
server.listen(8888);
```

Listing 3.6: Example of a simple echo server

Events can have any arbitrary string value as their key. The only reserved key is *error*, which is reserved for error events. The event listeners can also listen and emit error events. It should be kept in mind though, that if an error event is emitted and it has no listeners, the execution of application will be halted and a stack trace is printed to the console. [12]

3.3.3 Asynchronous Challenges

Asynchronous command flow brings challenges to the development of an Node application. The execution order of the code and the state of the application might not always be obvious or variables value might change unexpectedly. Listing 3.7 first defines a asynchronous function that will call its own callback after 500ms delay.

```
function asyncFunc(callback){
  setTimeout(callback, 500);
}

var one = 1;

asyncFunc(function() {
  console.log("One plus one is " + (one + one));
  // This is executed 500ms later
});

one = 2;
```

Listing 3.7: Example of challenges with asynchronous command flow

During this time the value of variable `one` is changed to 2. The console.log will output "One plus one is 4", which might not be what was expected.

Asynchronous command flow may also affect the completion of the application. The event-loop of Node keeps track of all asynchronous logic that has not yet completed and prevents the application from exiting. For example open database connections keep the application from exiting. This might be the desired outcome, e.g. for a web server, but not for some command line tool. [12]

3.4 Testing with Node and JavaScript

As applications grow in size and in the number of developers, it becomes harder and harder to keep track that everything works as it is supposed to. Because of this, automated testing has become an important part of any application development. Next, we will look into the automatic testing of Node applications. The asynchronous command flow of the Node brings challenges to testing. Developers need to take care that asynchronous unit tests that run in parallel do not interfere with each other. In this thesis unit testing will be covered with test-driven development (TDD) and behavior-driven development (BDD) models using Node's own assert module and third-party testing modules Mocha and Should.js.

3.4.1 Unit Testing with Node's Assert Module

The built-in assert module of Node is the basis for unit testing in Node. `Assert` command tests for a condition, and if the condition is not met, it throws an error. The assert module is also the basis of every third-party testing framework.

Assert module contains common functions for testing: `equal`, `notEqual`, `strictEqual`, `notStrictEqual`, `deepEqual`, `notDeepEqual` and `ok`. All these functions except `ok` take in three parameters: variable to test, value to test against, and an error message to show if the variable and the value differ. As `ok` function only tests for variable being `true`, it only takes in two parameters. `Equal` and `notEqual` functions use the more permissive version of the comparison operator (==). The *strict* versions use the stricter comparing operator (===). `Deep` versions compare two objects recursively. This means that if an object consists of other objects, also those objects will be compared for the equality. [14]

3.4.2 Unit Testing with Mocha and should.js

Mocha and should.js are popular third-party modules for unit testing. Mocha is a testing framework that is mainly used for BDD testing but it can also be used for TDD testing. Should.js module is used for assertions, it helps to describe assertions BDD style, which makes them more easy to understand. Should.js augments `Object.prototype` with a `should` property, which is used for assertions. Should property has many functions that make reading assertions simpler, e.g. `have`, `be`, `a`, and so on. These functions do not do anything, they just make the assertions more easy to read. Should.js is designed to be used with other testing frameworks, for example Mocha. [15]

Logic of Mocha tests are defined by a set of descriptive functions called `describe`, `it`, `before`, `after`, `beforeEach` and `afterEach`. Mocha also has TDD style equivalents for these functions but in this thesis we will concentrate in the BDD style functions. In Mocha tests, `describe` function is used to define a testing suite, which then can contain other `describe` functions and `it` functions. `It` function defines a single test to be executed. `It` function can take an optional callback parameter, which is used to define asynchronous tests. This callback is usually named `done`. `Before` and `after` functions are used to define logic that needs to run before or after tests, e.g. populate database. They both take a callback as an argument. `BeforeEach` and `afterEach` functions behave similarly to `before` and `after`, except they run before or after each test.[16]

Listing 3.8 presents a simple message list that can be used to push new messages into, delete all messages, get the amount of messages and a asynchronous function, which will get called after 1s delay. This function can be used to mimic a asynchronous database query.

```
function Messages () {
  this.messages = [];
};
Messages.prototype.add = function(message) {
  if(!message) throw new Error('No message specified');
  this.messages.push(message);
};
Messages.prototype.deleteAll = function() {
  this.messages = [];
};
Messages.prototype.amount = function() {
  return this.messages.length;
};
Messages.prototype.async = function(callback) {
  setTimeout(callback, 1000, true);
};
```

Listing 3.8: Simple message list module

Listing 3.9 illustrates how to use Mocha and should.js to write easy-to-read BDD style tests for the message module defined in Listing 3.8.

```
messages = new Messages();

describe('Messages module tests', function() {
  beforeEach(function() {
    messages.deleteAll();
  });
  it('should add items to messages', function() {
    messages.add('New message');
    var amount = messages.amount();
    amount.should.be.a.Number;
    amount.should.equal(1);
  });
  it('should do things asynchronously', function(done) {
    messages.async(function(){
      console.log('Saved something');
      done();
    });
  });
});
```

Listing 3.9: Testing with Mocha and should.js

First, a new suite is defined to test the message module. The first `it` function tests the `add` function of messages. One message is added to the array and then the amount of messages is checked. Second `it` function tests the asynchronous function and illustrates how to use the Mochas `done` callback.

4. SOFTWARE DESIGN

Web store product scraping is a complex process, which requires multiple different interconnecting software modules. These different modules are designed individually, and together they form the final scraping software. In this chapter we take a look at the different software development architectures and patterns that are used to develop the web store product scraping software. First, we take a brief overview of the software that is going to be implemented. Secondly, we talk about the architectural patterns that were used in the implementation. From the architectural patterns we first take a look at service-oriented architecture (SOA), which is very common amongst web based applications. Secondly, we talk about event-driven architecture (EDA) that is used in the communication between the different application modules. Third, we take a look at Template Method pattern. This pattern can be used in the creation of multiple similar modules, which share the same overall algorithmic functionality but have some individual modifications to the algorithm. After architectural patterns we take a look at how the implemented software is going to be tested with unit tests. In the end of the chapter we take a look at different database systems and evaluate their suitability for the web store product scraping software.

4.1 Software Overview

Web store product scraping software consists of three individual parts: a web site crawler, a product parser and a database. The web site crawler is a module that crawls the web stores for products. The crawler starts from the root page of the store and traverses systematically through the site, trying to find all web pages of the store. The crawler works by downloading the HTML of a web page and then finding all the links from it. It adds all the links to a process queue and continues processing the queue until it is empty.

After processing the HTML for its own needs, the crawler passes the HTML to the product parser. The product parser analyses the HTML and tries to find the HTML elements that contain important information about a certain product, e.g. price, name or description of the product. Finding of the important elements is based on a set of predefined rules. After the product parser has found all the necessary attributes for a product, the product is passed to the database. The database validates that the product is complete with all the necessary information and that the information is formatted correctly. If the product passes the validation, it will be saved to the database.

The crawler, the product parser and the database form a pipeline, which the HTML passes through, transforming the HTML to a final product in the database. There can be multiple separate product scrapers that each consist of one crawler and one product parser component. A single product scraper functions independently and parses one web store for products. All these separate modules communicate with the single database, which holds all the products. The structure of the software is pictured in Figure 4.1.

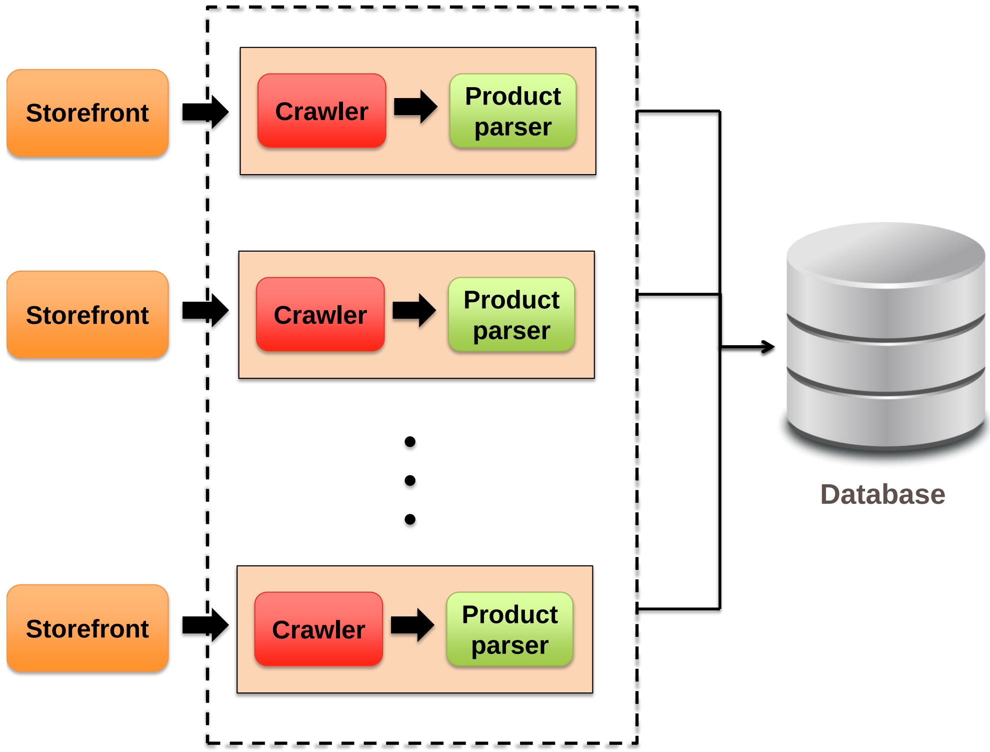

Figure 4.1: Overview of the web store product scraping software architect. The information flows from the storefront as a HTML file to the crawler. The crawler passes it to the product parser for data processing. In the end the information of the found product is stored to the database.

4.2 Architectural Patterns

Architectural patterns are general, reusable development patterns used in software development. These patterns provide good guidelines and solutions to commonly occurring design problems within a given context. Next, we take a look at some of the architectural patterns used in the implementation of the web store product scraping software. [17]

4.2.1 Service-Oriented Architecture

Different software components usually depend on the features of each others. These dependencies can be divided into real and artificial dependencies. A software has a real dependency, when it depends on the functionality provided by an other system. An artificial dependency in the other hand is a dependency that the system has in order to satisfy the real dependency. A real world example would be a travellers need for electricity, which is a real dependency. To reach this dependency, we have an artificial dependency: the need for the correct power plug to fit to the local power outlet. When the artificial dependencies between systems are reduced to the minimum, the systems are said to be loosely coupled. [18]

In Service-Oriented Architecture (SOA), the goal is to achieve a loose coupling between interacting software components. In SOA a software component can be a service provider, a service consumer or both. Providers offer services that consumers use to achieve the desired result. The result of the service usually lead to a change of state for the consumer and sometimes also for the provider. SOA achieves loose coupling between interacting software components by employing two important constraints:

1. All software components should have only a small set of simple and ubiquitous interfaces that only encode generic semantics. The interfaces should be universally available to all other providers and consumers.

2. Messages between interfaces should be descriptive with clearly defined extensible schema. A extensible schema allows an introduction of a new schema without breaking the existing services. Messages should not prescribe any behavioural information. [18]

There is usually only a few generic interfaces available in SOA. To achieve wide variety of functionality, application specific semantics must be expressed in the messages over the interfaces. To achieve a service oriented architecture the system must follow these rules:

1. The messages must be descriptive instead of instructive. The service provider is responsible of solving the problem and the service consumer is only interested in the outcome.

2. The messages should have a strict format, structure and vocabulary that all interested parties can understand.

3. The messages and the software system itself should be extensible to accommodate new features.

4. SOA software must have a feature that enables the service consumers to find the service providers. [18]

Figure 4.2 presents a diagram of SOA with a centralized service consumer and provider registry.

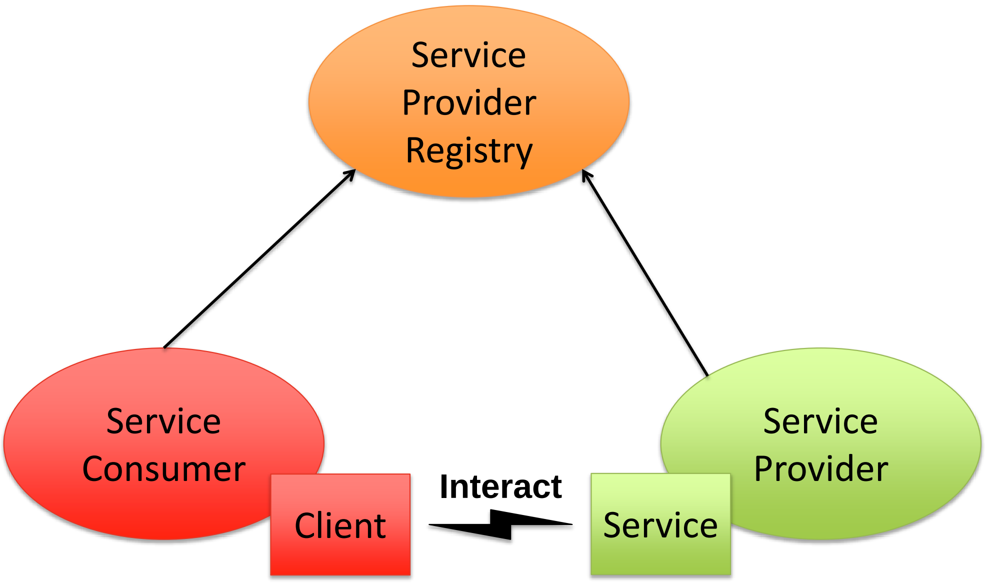

Figure 4.2: A diagram of SOA with a centralized service consumer and provider registry.

SOA is usually linked to big enterprise level systems that offer services for each other. A web based service is a common example of SOA. In a web service a web server offers an interface to serve web pages through an internet protocol that a browser, the consumer, consumes. [18]

SOA can also be used in smaller software projects. In the product scraping software, the crawler and the parser can both be thought as the service provider and consumer, and the database as a service consumer. The crawler provides an interface for transmitting the HTML files that it has requested from the internet. The parser has an interface for parsing the HTML to a product. The parser consumes the crawlers HTML and serves a product to the database. The database consumes the product by saving it to database. Distributing different services to own modules allows the software to scale with quite ease. There can be as many instances of each module as is needed.

4.2.2 Event Driven Architecture

From the point of view of Event Driven Architecture (EDA), an event is a notable thing that occurs inside or outside of the system. It can be a problem, an opportunity, a threshold, or a deviation. Each event should contain a header and a body. The header contains meta information about the event, e.g. event specific identification, type, name and timestamp. The event body should fully describe what happened so that all listeners can use the information without needing to know anything about the source of the event. [19]

In EDA, when a notable thing, an event, happens inside or outside the system, it immediately disseminates to all listeners. The listeners then evaluate the event and if needed, act on it. EDA is extremely loose coupled and usually also highly distributed. The source of the event only knows the event. It has no knowledge of the listeners of the event or the subsequent processing. EDA is best used for asynchronous flows of work and information. [19]

In SOA, a service composition might be constructed so that the service consumer is dependent upon an event in the service provider. E.g. in the web scraper the product parser depends on the web pages that the crawler downloads. Polling the service provider for the event would make the service composition inefficient and error prone. This polling pattern is pictured in Figure 4.3. [20]

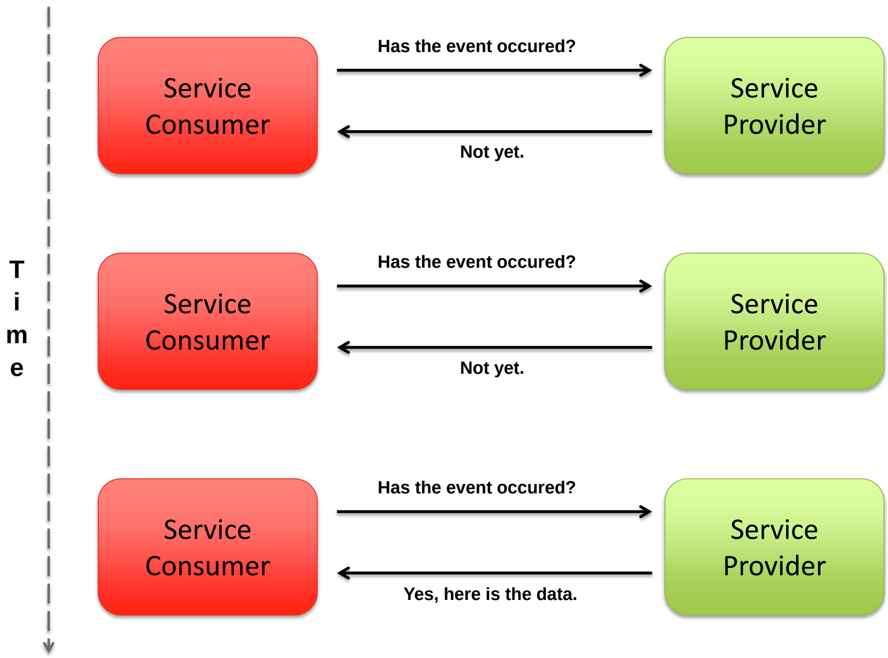

Figure 4.3: Event-driven polling pattern. The service consumer is constantly polling for new information from the service provider.

Usually the service consumer cannot poll the service provider in a secure way, which would guarantee that no events are missed. Polling also makes the service consumer directly dependant of the service provider. This increases the coupling between them and decreases the autonomy of the individual service components. Constant polling also consumes both the service consumers and service providers resources as they are exchanging unnecessary messages.

Event-driven messaging pattern is an improvement to the polling event-driven pattern. Event-driven messaging pattern is based on the Observer pattern. In the Observer pattern, service providers and consumers register themselves to an observer. When an event happens in the service provider, it notifies the observer, which then notifies all the interested parties. The use of an observer fully decouples the service consumer and the service provider. The observer also makes the behaviour of the service composition more predictable and reliable as it makes sure that the service consumer does not miss any events. The event-driven messaging pattern is pictured in Figure 4.4. In the middle is an event manager, or an observer, which disseminates the events between interested parties. [20]

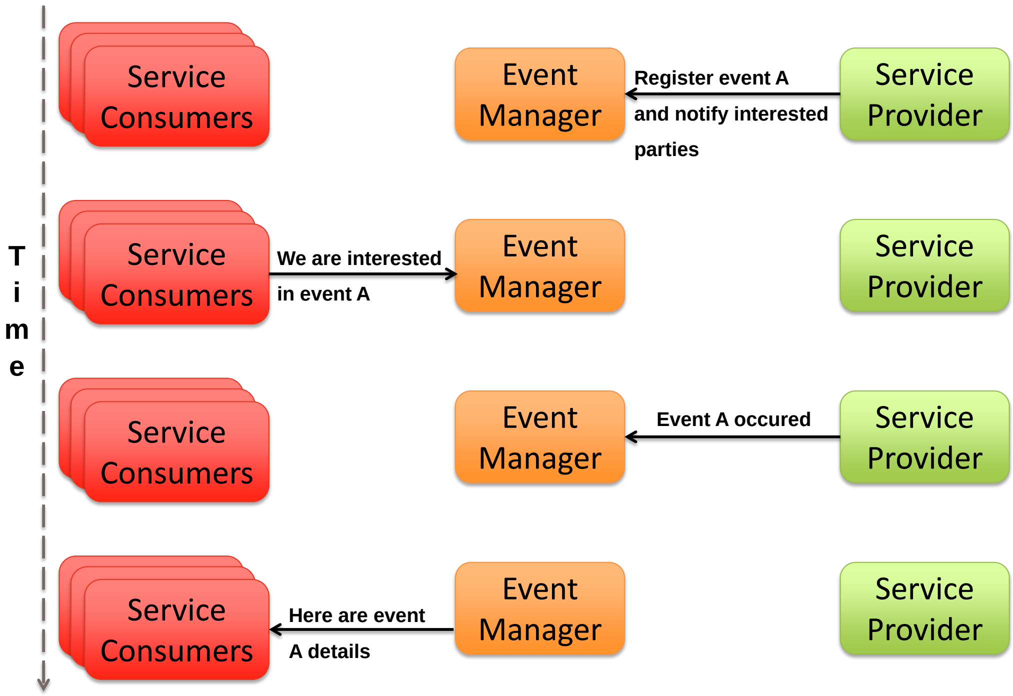

Figure 4.4: Event-driven messaging pattern. The service provider and consumer register themselves with an event manager. The event manager then takes care of relaying the events.

In the web store product scraping software, EDA is used to communicate between the different modules. When the crawler finds a new web page it fires an *pageFound* event, which has the actual HTML as the event body. After broadcasting the event, the crawler continues crawling the web page for new links and pages. The product page parser is listening for the *pageFound* event and catches it. The parser then starts to process the HTML and tries to find relevant information of the product. When the parser is done with the processing, it will fire an *productFound* event with the found product information as the event body. The database module is listening for the *productFound* event and will then save the new product to the database.

EDA is especially well suited for the web store product scraper as the different modules do not need to know about each other. After a module dispatches an event, it does not care about how the event and the data is processed further and the module can continue its own task. Also, as this architecture results in asynchronous product processing, the other modules will not slow down the crawler module, which is already the slowest process. This is mainly caused by the network latency and other slowing effects of the network.

Figure 4.5 represents a single scraper module. It consists of one crawler and one product parser. The HTML flows as an event from the crawler to the product parser and the final parsed product is stored in a database.

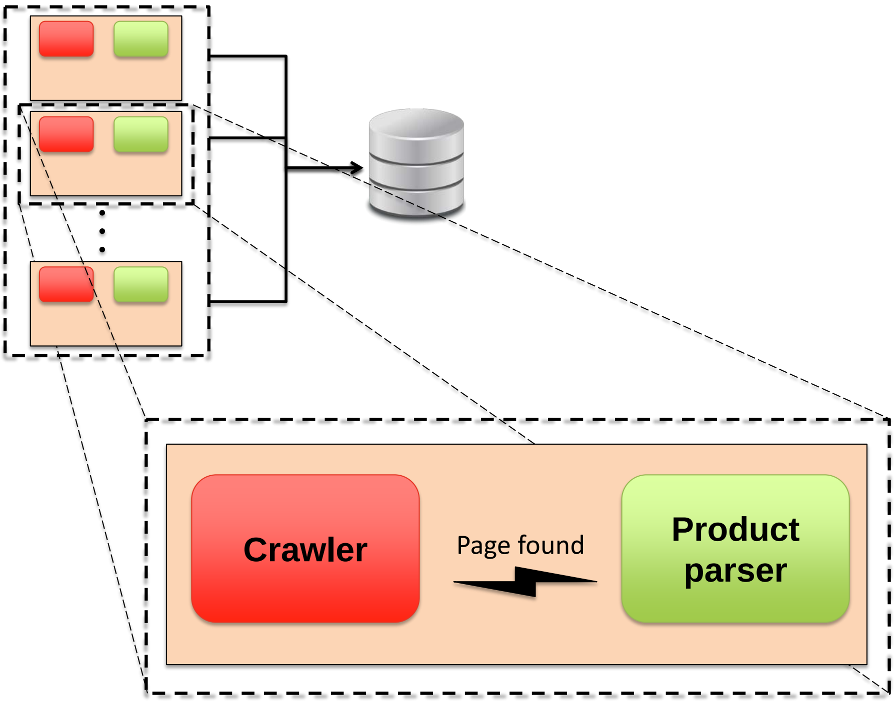

Figure 4.5: Single scraper module with one crawler and one product parser

4.2.3 Template Method Pattern

Template Method pattern is a behavioural design pattern used with the object-oriented programming and inheritance. In template Method pattern a class defines the skeleton of an algorithm. The class then defers some steps of the algorithm to its subclasses. This allows the subclasses to alter certain steps of the algorithm while keeping the overall structure the same. [17]

In practice, a base class is created first. This base class provides the basic steps of an algorithm. These steps are usually implemented as abstract methods. The subclasses then implement and change the abstract methods to create the wanted action. This way, the general algorithm is saved in one place, but the concrete steps may be changed in the subclasses. [17]

Figure 4.6 presents a simple class diagram.

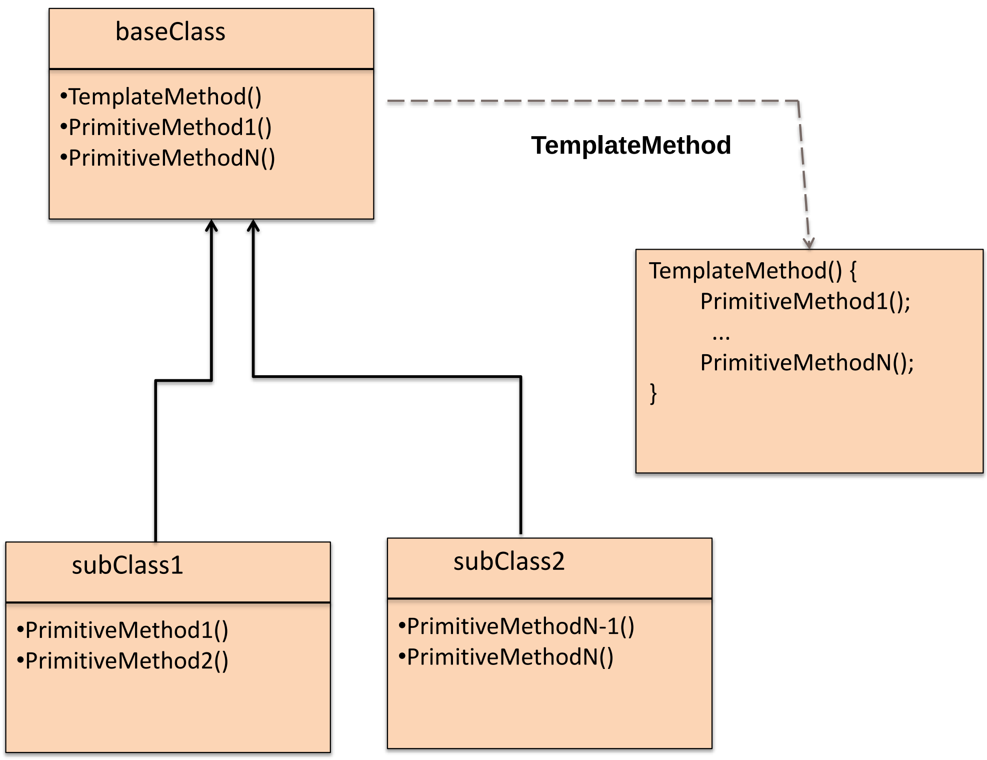

Figure 4.6: Visualization of a simple class hierarchy that implements the template Method pattern.

The base class holds the Template Method with the outline of the algorithm. The algorithm uses the primitive methods which can be altered in the subclasses. There are two subclasses inheriting the base class. They both have their own implementations of some of the primitive methods or all of them. The amount of alterations depends on the base class and the use case. The algorithm execution order stays the same in the subclasses but the outcome may vary.

In the web store product scraper, the Template Method pattern is especially useful in the product parser module. In the product parser module, we define a method with skeleton algorithm to find the right HTML element, extract correct value from it and then process the value to wanted format. As almost all webstores differ in some way, the Template Method pattern allows to tailor the different parts of the parser algorithm for each webstore. As shown in Figure 4.6 the product parser base class would have a skeleton of the parsing algorithm, which can then be inherited and modified to suit different web stores.

4.3 Testing

Testing is an important part of any software development project. As the web store product scraper will be continuously developed and configured to support more and more web stores. It is important that when supporting new stores the support for old stores will not be compromised.

Unit testing is well suited for repetitive testing of multiple small aspects of a software. Unit testing is a software testing method, where individual units of the software are tested with a specific set of control data, usage procedures and operating procedures. A unit is the smallest testable part of the application. It can test the entire module of the program, or more commonly an individual function or procedure. Ideally, each test case is independent from others. Sometimes it is also feasible to test that multiple consecutive tests work independently but also in conjunction.

Continuous unit testing has many benefits in software development. As new features are continuously added to the program, unit tests can guarantee that the existing features still work. If old features are not working, unit tests can help in finding the problem areas. Unit tests also make refactoring the old code more safe, as we can be sure that the program works same way as before refactoring.

The web store product scraper is thoroughly unit tested throughout the development process to ensure correct product parsing. Both, the crawler and the parser, are tested individually as well as a complete module. The parser will be tested on every significant development milestone. It is not reasonable to ensure with unit tests that every individual web store works with the parser as there will be hundreds of different stores. Instead it is more beneficial to test that a small set of stores that represent a certain feature improvement works correctly.

4.4 Database Systems

Traditionally, Relational Database Managing Systems (RDBMS) have been the choice of database for many systems since the 1980's. In relational databases, the data is presented as relations, a tabular form which consists of a collection of tables. Each table consists of a set of rows and columns. A relational database is usually managed through Structured Query Language (SQL).

The rise of big data and real-time web applications have increased the need for new database systems. Not only SQL (NoSQL) is a term used to refer these non-relational database systems. In NoSQL the data is modelled in other means than the tabular schema of relational database, e.g. in documents or graphs.

Next, we will look into these two architectures more thoroughly to determine which one suites better the needs of the web store product scraper.

4.4.1 RDBMS and SQL

In 1970, an IBM employee Edgar Codd published a paper called "A relational Model of Data for large shared Data Banks" [21]. This paper introduced the basic concepts of a relational database systems:

- The databases internal representation should be independent of the hardware or software configurations of the system.
- A high level non-procedural language should be used to manipulate the database.
- The concept of relations, primary and secondary keys, and logical operations, which are used to manipulate the database.

A relation is a set of tuples with the same attributes. A single tuple usually represents a single object with a set of individual information. Objects typically represent physical objects or concepts, e.g. employees or blog posts. A relation is usually described as a table with rows representing tuples and columns representing the attributes of tuples. Figure 4.7 presents the relational model of a relational database. A relation consists of tuples, which consist of attributes. The attributes are the same across tuples in a single relation. [21]

Tuples by definition are unique and their attributes constitute to a superkey that can be used to identify the tuple. Using a superkey constituted of all attributes can be troublesome when dealing with a lot of attributes. Because of this, tuples can also have a primary and a secondary key to help to identify tuples. The primary and secondary keys, or combination of them, are unique across a single relation and can be used to easily identify tuples. The relational model states that the tuples or their attributes are not in any order. Instead, the order and access to the specific data is specified through queries that select and order the specific set of tuples. [21]

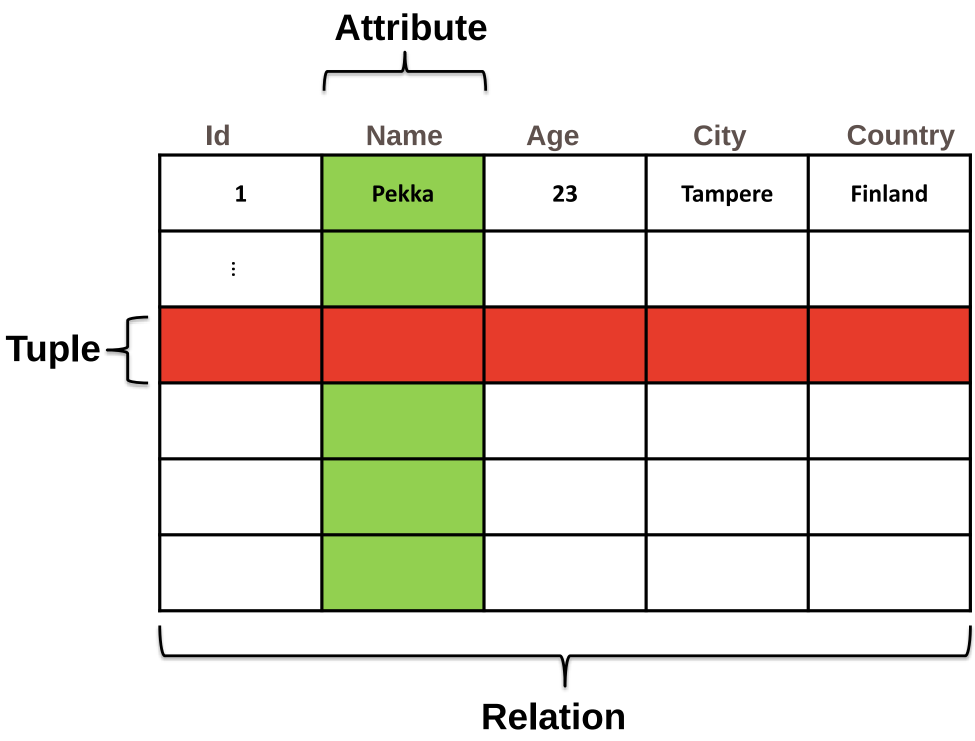

Figure 4.7: Relational database terminology. Relation represents the whole table. Tuple is a single row in it. An attribute represents a single value in a tuple and together they constitute to a column in the table.

A set of database commands are called a transaction. Transaction is a single unit of work in the database management system. It allows the correct recovery on failures and can be used to track changes in the database. Relational databases usually implement ACID (Atomicity, Consistency, Isolation, Durability) properties in their transactions:

- **Atomicity** requires that every part of a transaction occurs or none of it. If one part of transaction fails, the database returns to a state before the transaction started.

- **Consistency** requires that a transaction will bring the database from one valid state to another. This means that all written data is valid according to defined database rules.

- **Isolation** requires that concurrent execution of transactions leads to same outcome as if the same transactions were executed serially.

- **Durability** requires that when a transaction has been committed the result is permanent and will be in place even if the database crashes immediately after commit. [22]

SQL is a example of standardized query language used to manipulate the relations in a relational database. SQL consists of a data definition language and data manipulation language. SQL enables data insert, query, update and delete operations, relation schema creation and modification and data access control. [23]

4.4.2 NoSQL

Non-relational databases have been around as long as relational databases. However, the term NoSQL was first used in 1998 by Carlo Strozzi to name his lightweight open-source non-relational database. The term was reintroduced in 2009 by Eric Evans in an event about open-source distributed databases. Since then, the term has been used to refer non-relational database systems. [24]

The main idea behind NoSQL is to provide a mechanism to model and store data in other ways than in a form of relational database. The data models are usually more permissive for differences between elements, unlike in relational database where every element in a table has the same attributes. In NoSQL there is no concept of table or column in the same meaning as in the relational data model. The goal of the NoSQL is to provide simpler design, better horizontal scaling, and finer control than relational databases. Though, this might sacrifice some availability. NoSQL databases usually implement the CAP (Consistency, Availability, Partition tolerance) theorem instead of ACID. CAP theorem translates that for a distributed computer system it is impossible to simultaneously provide all of the following principles:

- **Consistency**; all nodes of the system see the same data at the same time.
- **Availability**; every request to the system receives a response, either success or failure.
- **Partition tolerance**; the system continues functioning despite arbitrary message loss or failure of part of the system.

Usually NoSQL databases provide two out of these three principles. [25]

NoSQL databases can be divided into many categories and subcategories by how they represent the data. Different data models and implementations optimize different aspects of the database and CAP theorem. Column, Document, Key-value and Graph data models are some of the most used data types for NoSQL databases. [25]

Column Data Model consists of tuples with three elements: unique name, value, and timestamp. Timestamp is used to determine which of the backup nodes are up-to-date. In relational database a column was part of the table and every row had the same columns. In the Column Data Model, the concept of table does not exist but a column can still be part of a column family. A column family can then form a similar concept as a tuple in the relational database to provide some order and hierarchy to the data model. The column families are independent from each others so there is no guarantee that if a column exists in one family it will also exist in others. Columns can also have a different order and meaning between families. [25]

The central concept of **Document Data Model** is a *document*. Generally a document encapsulates and encodes data in some standard format. What this format is, differs between implementations, but some popular formats are XML, JSON, BSON (Binary JSON), and YAML (YAML Ain't Markup Language). Compared to a relational database, a document forms a tuple in the database. Documents are addressed in the database with an individual key. The key can be human readable or some hash key, but it has to be unique. The documents can also form a collection of documents. These collections would then form a similar concept to relational table. In the Document Data Model, each individual document in the collection can have completely different fields of data, and it is the responsibility of the user to keep the database organized. [25]

Key-value Data Model is one of the simplest NoSQL data models. It uses a map or hash table as the fundamental data model. The data is represented as a collection of simple key-value pairs, so that every key is unique in the collection. Key-value Data Model has many different subcategories according to different attributes of the database, e.g. consistency of the data model, order of the keys and data storage solution. [25]

Graph Data Model is based on the graph theory. It uses *nodes*, *edges* and *properties* to represent the data. The *nodes* represent entities such as student or employee. The *properties* represent information about a single node, e.g. student number, name or status. *Edges* connect the nodes to other nodes or to properties. Edges can also have their own meta information. In graph databases, the most important information is usually stored in the edges. The edges can then be used to reveal meaningful patterns between nodes and properties. Graph databases are normally used for information, where the meaning lies in the connections of the nodes or if graph theory queries are needed, for example for finding the shortest path between two nodes. Graph databases are normally slower than other NoSQL data models with operations that modify a large set of elements in a similar way. Normal relational database type queries, like ”find all students with status active”, are also slower with graph databases. [25]

4.4.3 Relational Databases Versus NoSQL Databases

The main differences between the two alternate database management technologies are: performance and flexibility. NoSQL databases generally process data faster in update and look-up intensive transactions. NoSQL databases also usually scale horizontally better than RDBMS. RDBMS instead process data more precisely and more safely due the ACID transactions. In addition of making the database faster, the simpler data model of NoSQL makes it also more flexible than rigid relational schema. As both RDBMS and NoSQL have their strengths and weaknesses, the choice of database system depends heavily on the situation. [26]

4.4.4 Database Requirements for Web Store Product Scraper

In the web store product scraper, the data model needs to be flexible. The products and store specific configurations vary in their information, and the database schemas can change multiple times due the agile development. Fast data processing is also more important criteria than reliability. Because of these aspects, a NoSQL database with Document Data Model was chosen for web store product scraper. NoSQL databases also often employ easier integration with JavaScript than relational databases. This is usually due the usage of JSON or BSON as the base data type of Document Data Model.

Database usage is continuous as products will be read or written to database almost constantly. Web store product scrapers will scrape web stores constantly with multiple different instances, thus writing to the database is parallel and continuous. Reading from the database will not be as frequent as writing and it will concentrate to times that someone is researching the data.

The database should be able to handle all write requests from every web store product scraper, as it is important to not lose a single product. A small read request latency is acceptable, to ensure that the writing performance of the database will not suffer. Heaviest operations performed for the database will be the searches through the whole product catalogue. Space requirements for the database are moderate as a single product will not take much space. However, the database will contain millions of products.

5. SOFTWARE IMPLEMENTATION

In the previous chapter we took a look on the architecture and design principles used to develop the web store product scraper. In this chapter we discus more about the implementation of the software. Figure 5.1 presents the structure of the software that will be implemented, and the flow of data between the different modules of the software.

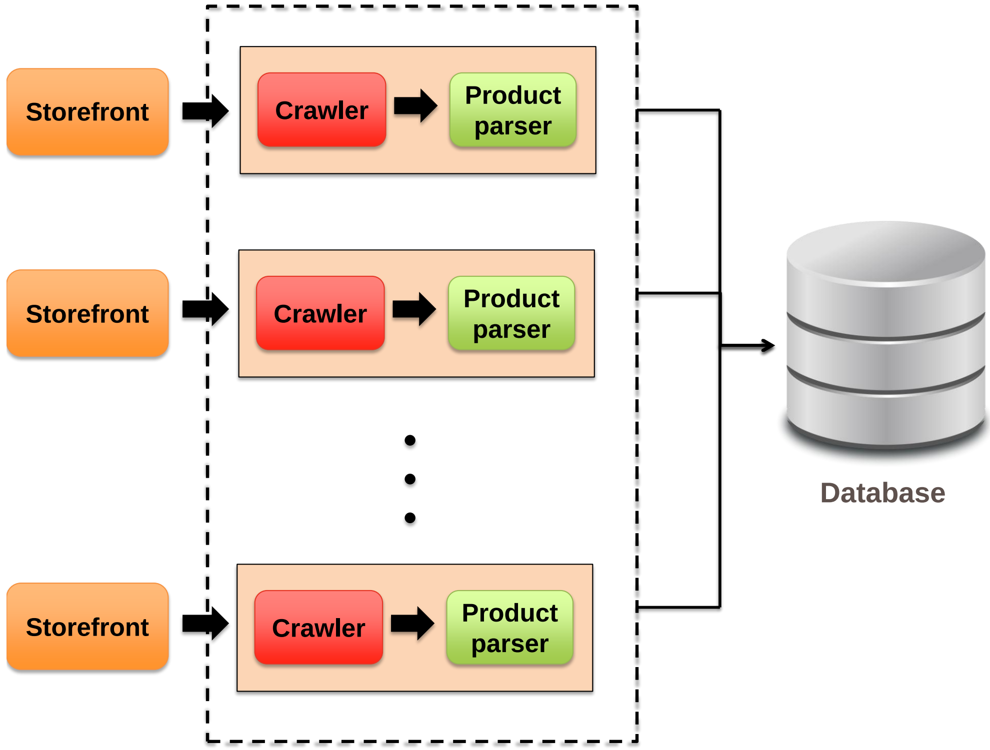

Figure 5.1: Overview of the scraper software architect. The information flows from the storefront as a HTML file to the crawler, which passes it to the product parser for processing and in the end the information of found product is stored to the database.

5.1 Web Store Crawler

The Web store crawler finds and retrieves the data for the other modules to process. A web crawler is a piece of software that methodologically downloads websites that are linked together. Web crawlers have been around since the dawn of the internet, analysing web sites for meaningful data. For example, Google constantly crawls the internet to enhance its search results. Crawlers operating logic is pretty simple:

1. The crawler is started on a single web page, e.g. www.tut.fi.
2. The crawler downloads the HTML code of the web page from the web server.
3. The HTML code is analysed for link elements, i.e. every <a> element of the HTML. From the <a> elements the value of `href` attribute is extracted. This contains the target of the link.
4. All `href` attributes are added to a data structure. The data structure is kept clean of potential duplicates so that the crawling comes to an end eventually.
5. Then the next item from the data structure is taken and the process starts from the beginning.

Figure 5.2 presents the operation flow of a web crawler.

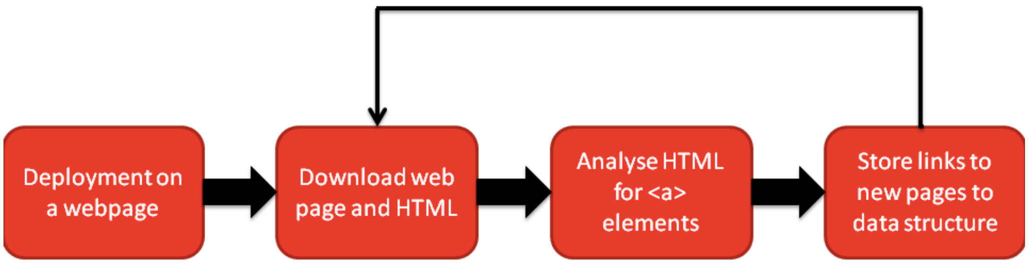

Figure 5.2: Operational flow of a web crawler

Web crawlers have been around for a long time, so there are many third-party implementations for web crawlers. As the software is developed in an agile environment, it is important to develop a working version as fast as possible. Because of this, a third-party crawler was used for the first version of the web store product scraper. After research, the Simplecrawler [27] was chosen as the crawler because of its features.

Simplecrawler is a simple web crawler that utilizes the EventEmitter class from Node.js. All important messages from the crawler are accessible and handled through events. The Simplecrawler is also extensively configurable to suit the needs of different web stores. Simplecrawler can run concurrently on a website to crawl it faster. The web pages do not depend on each other, so they can be fetched concurrently as long as every concurrent download utilizes the same data structure for the links.

This ensures that every page is downloaded only once. The destination webserver can throttle a single connection but many simultaneous connections can bypass this setback. So even if a single connection is throttled down, the overall speed of the data collection is faster with concurrent downloading.

When Simplecrawler has downloaded a web page, it first analyses the HTML as described in the previous section. After that, it fires a *fetchcomplete* event. In the handler of the event the URL of the web page is matched against a store specific pattern. This is done to determine, whether the web page should contain information about a single product. Many web stores have an URL schema in which a page containing a product has an URL with pattern "*/product/*". This can be used to optimize some web stores to evade unnecessary page parsing. Figure 5.3 presents the flowchart of Simplecrawler.

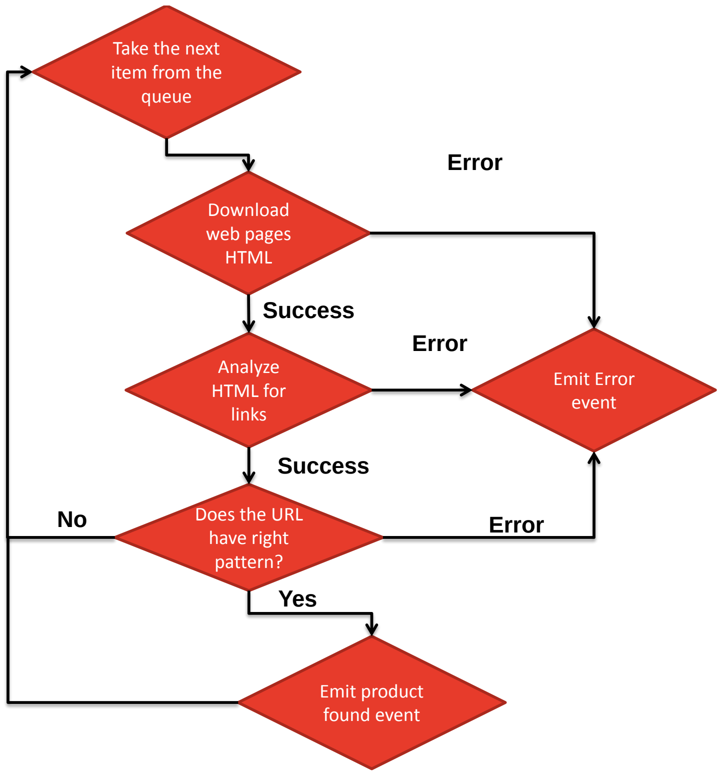

Figure 5.3: The flowchart of Simplecrawler

Simplecrawler has a wide collection of different events it emits during the crawl. This is especially useful in the case of error events. Different error events can be handled differently depending on the page in question. If the homepage of a web store emits an error, it is then knwon that the target web server is not responding or our configurations are wrong. Either way, in response to this error, the web store can be disabled until further investigation for the source of the error has been done. This way, unnecessary crawls that are known to fail can be avoided. On the other hand, if a random web page in a web store throws an error, it only affects the single page, not the whole store. This error can be distinguished from other errors and logged as a warning without afecting the crawling.

5.2 Product Parser

When the web store crawler finds a product, it will emit a *productFound* event with the HTML body of the web page. This event will be handled by the product parser, which will extract the valuable product information from the HTML code. Parsing the HTML code is based on a store specific settings that specify how to extract the meaningful information for each product attribute. Parsing the HTML code is based on the Document Object Model (DOM) API and the use of Cascading Style Sheet (CSS) selectors to select the valuable HTML elements.

5.2.1 Document Object Model

DOM is a specification released by W3C. This specification specifies a set of standardized programming interfaces for working with structured documents, e.g. XML or HTML. DOM standard is programming language neutral and today it has libraries for almost every popular programming language, including JavaScript, Java, C/C++, Python. [28]

DOM is an object model, where the structure of a document is modelled with *objects*. The objects describe the structure and behaviour of the elements in a document. DOM is usually represented as a *tree* structure where the elements of the document are its *nodes*. Listing 5.1 shows a simple HTML document.

```
<html>
  <head>
    <title>This is a document.</title>
  </head>
  <body>
    <p>This is some text!</p>
  </body>
</html>
```

Listing 5.1: Simple HTML document

The above HTML can be illustrated as a simple tree structure. This tree is pictured in Figure 5.4

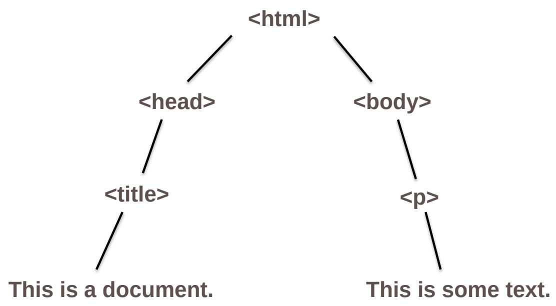

Figure 5.4: The DOM tree of Listing 5.1

The DOM tree in Figure 5.4 has the `<html>` tag as its *root* node. The `<head>` and `<body>` would be the *children* of the root node and thus each others *sibling* nodes. `<Title>` and `<p>` tags would again be the *children* of their *parent* nodes, but they are not *siblings* as they have different *parents*. Note that also `<title>` and `<p>` tags have a *child* node called `text` node that holds the text information of those nodes. [28]

DOM specification defines an API, which is used to modify and work with structured documents. The API provides important methods to create and modify the structure of the document, traverse in the document, and to attach events to documents objects, e.g. mouse over or mouse click events. All these functionalities are important in a modern web site development with HTML and JavaScript. For the product parser module, the most important functionality in DOM is traversing it with CSS selectors. [28]

5.2.2 Cascading Style Sheet Selectors

CSS is a standard maintained by W3C for styling a HTML document. CSS provides a set of rules that are used to style the HTML document. As HTML only describes the structure of the document, CSS describes the styling of it. To allow styling of the HTML elements, CSS needs a set of rules according to which the right elements can be identified and styled. Listing 5.2 illustrates some simple CSS rules. CSS rules consist of a selector and a set of properties and their values. Listing 5.2 defines that all `<h1>` elements are in red with font size of 15 pixels and that all elements with class `big` are 500px wide and have a margin of 10px around them. [29]

```
h1 {color: red; font-size: 15px}
.big {width: 500px; margin: 10px}
```

Listing 5.2: Example of CSS

The selectors that define, which elements are styled, are called CSS selectors. As these selectors define a way to easily select HTML elements according to each elements attributes, they are essential for product parser to identify the right elements. The correct elements can also be identified by traversing the DOM tree with DOM API, e.g. by selecting the third child of the root. However, this would be much more laborious than a simple CSS selector.

CSS selectors can select HTML elements according to their tags, attributes or hierarchy in the DOM tree. The most important CSS selectors for product parser are illustrated in Table 5.1.

Selector	Example	Example description
.class	.category	Selects all elements with class="category".
#id	#brand	Selects the element with id="brand".
*	*	Selects all elements.
element	p	Selects all <p> elements.
element,element	div,p	Selects all <div> elements and all <p> elements.
element element	div p	Selects all <p> elements inside <div> elements.
element>element	div>p	Selects all <p> elements where the parent is a <div> element.
[attribute]	[itemprop]	Selects all elements with a itemprop attribute.
[attribute =value]	[itemprop =name]	Selects all elements with a itemprop attribute containing the word "name".
[attribute^=value]	a[src^="https"]	Selects every <a> element whose src attribute value begins with "https".
[attribute$=value]	a[src$=".jpg"]	Selects every <a> element whose src attribute value ends with ".jpg".

Table 5.1: A set of CSS selectors

Using these selectors and a couple of additional settings on how to process the selected elements, it is possible to parse a whole web page for a single product.

5.2.3 Implementation

The product parser module was implemented as a class with a Template Method for parsing a single product from the HTML code. The Template Method consists of the following tasks:

1. Build a DOM tree from the HTML code.

2. For each property of a product, extract the right HTML element from the DOM tree with a CSS selector.

3. Extract the right information from the text node or from a certain attribute of an element.

4. Process the extracted value by removing the unneeded information with regular expressions and trim the white space.

The product parser class has separate Template Method algorithms for properties with a single value or an array of values. For example, the brand of a product is usually a single value, but the images and categories of a product usually consist of multiple values. Both of these methods function as described previously but the outcome depends from the method. The multiple value method will always return an array, even if its only a single value long. After the properties of a single product are processed, the product is validated and passed on to the database.

5.3 Database

The database of the product parser was decided to be implemented with NoSQL database. After researching multiple options, MongoDB was selected. MongoDB is an open-source document database and according to their homepage "the leading NoSQL database". [30]

5.3.1 MongoDB

The main development principle of MongoDB was to design a relational database, but switch the data model to *document* based NoSQL. Because of this design philosophy, MongoDB supports indexes, dynamic queries and fast updates like a relational database. These features are essential for the web store product scraper as the data schema of the product can change rapidly during the agile development. Products are also constantly scraped and searched, so indexes and fast updates are important. MongoDB also uses JavaScript as its API and query language so it is easy to integrate to Node program. [30]

MongoDB combines saved documents to collections that are then saved to the database. A single MongoDB deployment can hold multiple databases and each database can hold multiple collections. [30]

A collection is a set of similar documents, and it is equivalent to a table in a RDBMS. As collections do not enforce any schema to *documents*, the *documents* in a collection can have different fields and the common fields can have different data and meaning. Collections are just a hierarchical system to group documents with related purpose together. [30]

Documents are key-value pairs. In MongoDB documents are analogues to JSON objects, which makes them especially well suited to operate with Node programs. Internally MongoDB saves documents in BSON format, which is a binary representation of JSON. BSON is a more efficient compared to JSON and it also has more available data types than a JSON structure. [30]

5.3.2 MongooseJS

Even though MongoDB uses JavaScript as its API language, it is easier to use MongoDB through a maintained and tested third party library. In the web store product scraper, a Node library called Mongoose is used. Mongoose acts as an Object Data Manager (ODM) between Node and MongoDB. Typically ODM provides an API to handle all the interactions between a database and the user. Mongoose provides an API for building of collection and document, type casting, validation and other business logic between JavaScript and MongoDB. [31]

In Mongoose, *documents* are based on *schemas*. A schema is a blueprint for the document and it also defines a single MongoDB collection. A schema defines what key-value pairs a document can have and also what kind of validation should be done for the document before saving it to the database. Schemas can also attach customized getters and setters for the documents or individual document values, e.g. there can be different getters for different date formats. [31]

Mongoose adds restrictions to document creation, as a document can not have properties that are not in the schema of the document or the value can not be of a different type than in the schema. The Mongoose schemas also provide document validation on a property level to ensure the correctness of the inserted documents. MongoDB itself does not offer any validation as the NoSQL database does not restrict document models. By working with Mongoose, it can be guaranteed that the documents are validated and congruent. Validation is especially important with the web store product scraper as the products are gathered from multiple different sources, and all of them must have a similar structure and validated values. [31]

5.3.3 Implementation

In the web store product scraper, two main Mongoose schemas were made: one for the products, and another for the web store specific scraping and parsing configurations.

Each product will have mandatory and optional attributes. The required parameters of a product are: a unique store identifier, a store name, a name of the product, an array of categories where it belongs to in the original store, a retrieval date, a price with value and currency, and an URL where the product was retrieved from. In addition to these, a product can also have a notion about its brand, an array of links to images, a description, and an array of other ids. Other ids can for example be Stock Keeping Unit (SKU) or International Article Number (EAN) codes. SKU is a store specific id for managing the inventory of the store. EAN is an international product id that should be unique across all stores.

Web store specific configurations will have settings for the basic attributes of a store, crawling, and product parsing. The basic setting of a store will consist of a name of the store and a link to the homepage of the store. The store name is unique across the database and the link to the homepage acts as a starting point for the crawling.

The crawling configurations of a store have settings to optimize the crawling. It has settings to limit the crawled pages and to filter the emitted product pages according to their URL. These can be used to steer the crawler away from pages that do not contain any information about products, e.g. the *about* pages of a store. This also optimizes the product parser as it will not try to parse pages that do not contain products. The crawler settings also have attributes to control the concurrency and the interval of consecutive page fetches. These are important settings to prevent fetch timeouts, by limiting the speed of crawling in stores that have some kind of speed limiters.

The product parsing settings of a store are a blueprint for the product parser about how to extract the attributes of a product from the HTML code. The HTML code of a product varies a lot from store to store, so also the parsing blueprints vary a lot. A product has two main categories for the value of an attribute: a single value or an array of values. Both of these can have a fixed value or the value can be parsed from the HTML. If the attribute is optional, the setting can also be empty. If the value is parsed from the HTML code, the settings will include a CSS selector for the correct element, a notion if the value can be found among the text of an element or from the attributes of the element (e.g. `href` attribute). The parsing settings also include settings for how to process the extracted value: should the value be parsed with a regular expression or should something be added to the value. For example the protocol and domain can be added to the image links, which are usually relative links. In the array formatted values, the settings can also include an option to limit the length of the array or which of the elements are selected. For example some web stores have the product itself as the last category, which can be safely ignored.

6. EVALUATION

The web store product scraper was tested by configuring it for three different sized store: a small, a medium and a large sized store. Each store was scraped seven times with different crawler interval and concurrency settings to determine the most suitable crawler settings and to evaluate the overall performance of the web store product scraper.

The tested combinations were 0 ms interval with one, five and ten concurrent scrapers, 500 ms interval with five and ten concurrent scrapers, and 1000 ms interval with five and ten concurrent scrapers. The different interval settings were selected to determine whether the stores used any network throttling. The different concurrency settings were selected to determine how well the web store product scraper scales.

6.1 Configuring Web Store Product scraper

On each store the crawler settings were optimised by running multiple crawls and setting the filters for all unnecessary pages. The product parsing settings on each store were configured with the help of a browsers DOM inspector to determine the right CSS selectors. Additional settings were added by hand and validated by running test parsings for a test product page.

Listing 6.1 presents one example of the product parsing settings of a single store. Every attribute of a product has multiple settings. The `attr` property tells the parser the name of the HTML attribute the value can be found. For example the image links can be found from the `src` attribute. The `selector` property specifies the CSS selector to select the correct HTML element. The `replace` property specifies the regular expression settings that are performed to the extracted product attribute. The `slice` and `index` properties specify how to process possible arrays of HTML elements obtained with the CSS selector. The `fixedValue` property specifies that the product attribute should have a fixed value instead of extracting it from the HTML. The `parse` property specifies a regular expression used to process the text obtained from a HTML element.

```
{
    "images": {
        "attr": "src",
        "selector": "#CurrentProductImage > img",
        "replace": {
            "what": "^",
            "with": "//www.example.fi" }},
    "categories": {
        "selector": ".Breadcrumb a",
        "slice": {
            "begin": 1,
            "end": -1 }},
    "storeId": {
        "selector": "[name$='[product_id]']",
        "attr": "value",
        "index": 0 },
    "description": {
        "selector": ".pc-text > p" },
    "name": {
        "selector": ".pc-text > h1" },
    "priceCurrency": {
        "fixedValue": "EUR" },
    "priceValue": {
        "selector": ".home-pb-price",
        "parse": "\\d+,\\d+" }
};
```

Listing 6.1: An example of product parsing settings of a single store

Using these settings the product parser can process the HTML code and extract the attributes of a product. For example, the product images can be found from the `src` attribute of a `img` element, which is a child of an element with `id` of "CurrentProductImage". The obtained text should then be processed by adding "//www.example.fi" to the beginning of it.

After each store was configured and the filtering for unnecessary pages was optimised, each store was scraped seven times with different crawl interval and concurrency settings. The results of these seven scrapes can be seen in the following sections. The measured attributes were the amount of products found, time taken and the amount of encountered errors (e.g. HTTP errors or connection timeouts). The relevant interval and concurrency settings are also shown in the figures.

6.2 A Large Sized Store

Figure 6.1 presents the results of seven test scrapes on the large sized store. From the large sized store approximately 23000 products were scraped on each scrape.

The amount of errors does not stay as stable as it seems to somewhat depend on the interval and the concurrency settings. It is good to note that the amount of errors is still significantly lower than the amount of scraped products. The amount of errors probably depend more on the interval setting than the concurrency. If the scraper requests for new data continuously with 0 ms interval, it strains the receiving web server more and the web server is more likely to respond with an error message. The server has harder time to answer each connection and the amount of timeouts rises. From the time consumed curve can be seen, that even if the interval is risen to 500 ms or even to 1000 ms the scraping time does not rise very dramatically. This is logical, as web servers usually scale better horizontally, which means that they can handle multiple simultaneous connections more easily. If we take a closer look at the measurements three and five, which are made with 10 concurrent scrapes and 0 ms and 500 ms interval. The time consumed on third measurement is not significantly higher even though there is a 500 ms pause between each new request. This might indicate that the receiving web server does some network throttling, which overtakes the slowing effect of the interval setting. Because of this, an 500 ms interval setting would be better for this store to reduce the caused unnecessary stress on both parties.

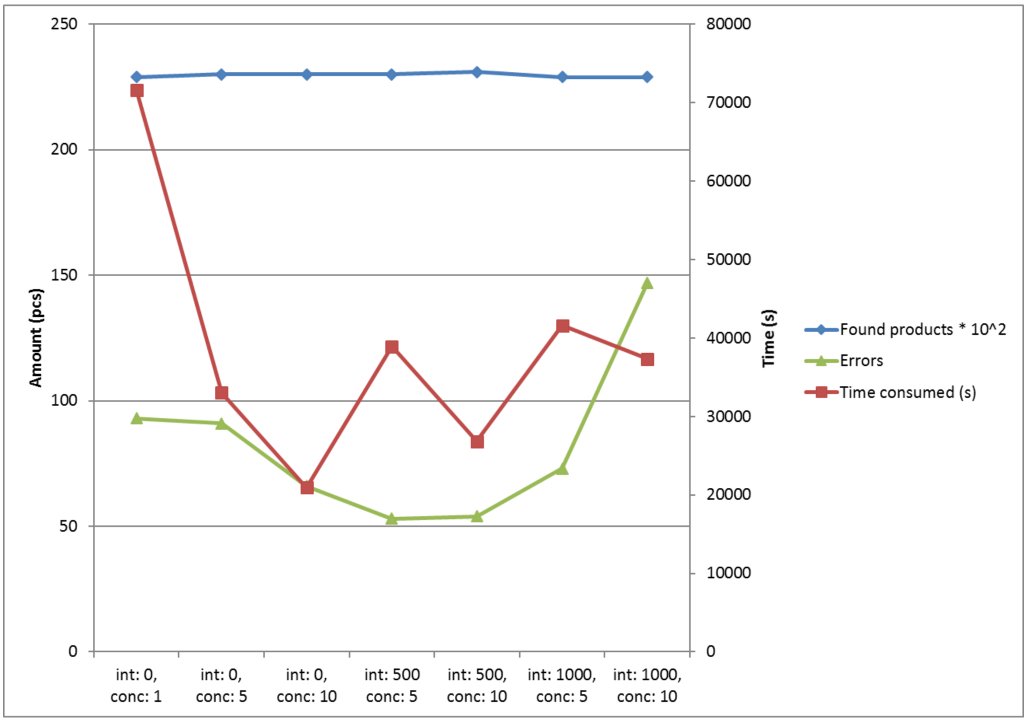

Figure 6.1: A large sized store

6.3 A Medium Sized Store

Figure 6.2 presents the results of seven test scrapes on the medium sized store. From the medium sized store the amount of products scraped was approximately 2700. The amount of errors does not seem to correlate with crawling settings as it did in the large sized store. Instead, the amount of error stays quite stable. This indicates that most of the errors are HTTP errors (e.g. 404 - Page Not Found), from broken links on the web page. These errors will be always encountered and should be avoided by fixing the filtering settings of the crawler. These errors can be a problem in the underlying product management software or the store database and could thus be fixed overtime by the store. In the medium store case the evidences of network traffic throttling is even more obvious as the different interval settings have even smaller effect than in the large store. Again a suitable interval time would be somewhere closer to 500 ms than 0 ms in order to avoid overloading the server.

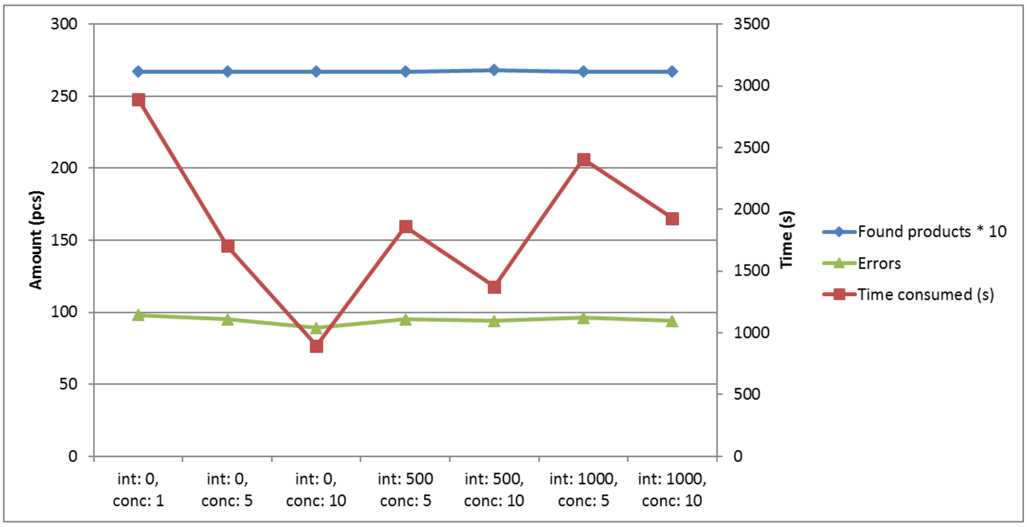

Figure 6.2: A medium sized store

6.4 A Small Sized Store

Figure 6.3 presents the results of seven test scrapes on the small sized store. From the small store the amount of scraped products was approximately 160. Also in this case, the number of errors seems to stay quite level. This indicates the same kind of results as with the medium sized store. In the small sized store it is clear that there is no network throttling, as the speed of the scrape seems to depend mainly on the interval setting. This might also be caused from the small amount of the web pages in the store. The effect of interval setting is not being overtaken by the large amount of product pages as it did with the large store. In these kind of stores the interval can be set to low and concurrency to high value as the receiving server seems to be able to quickly answer all our requests and the amount of web pages is so low that the caused stress is only temporally.

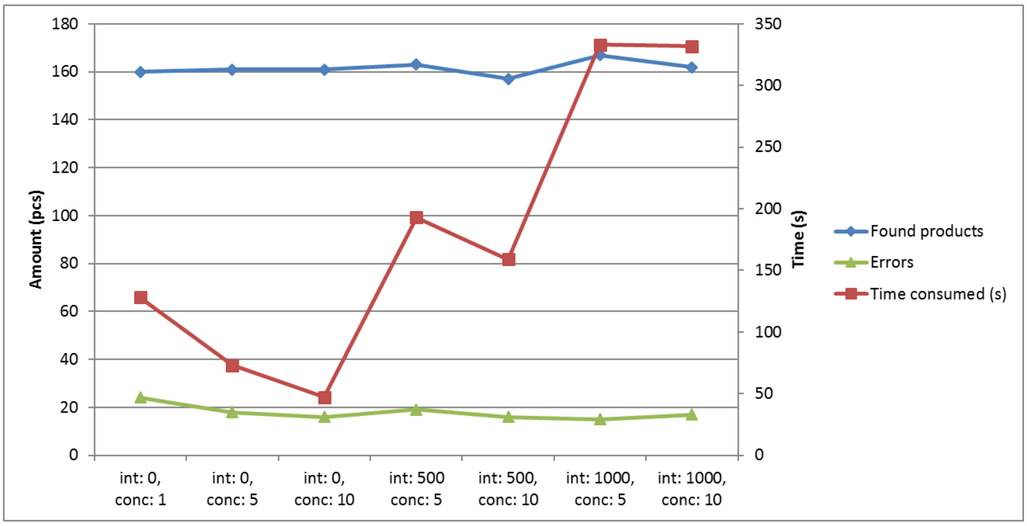

Figure 6.3: A small sized store

As can be seen from the test cases, the time it takes to scrape a stores depends heavily on the size of the store. Bigger stores take a longer time to scrape because those have larger amount of products and similarly larger amount of HTML files to fetch. The amount of collected products is quite level across all scrapes. This indicates that on each scrape the scraper finds the same pages. Also almost every product from a store can be acquired with each scrape.

Usually stores do not update their product catalogues or prices very often. Scrapes can thus be performed once a day at most or even once a week. The scraper can also cause a lot of stress for the network server of the store, which should be taken into account. The interval setting of the crawler should be tuned in accordance with the network limiter of the store. Scrapes should also be concentrated to night time when the stores have a lower amount of other network traffic.

The web store product crawler worked out successfully. The selected programming architecture patterns turned out to be right choices for the implementation. As can be seen from the tests, the performance and the configurability of the web store crawler is excellent. The web store crawler also scales nicely even to bigger stores as multiple crawlers and parsers can be executed concurrently.

6.5 Future work

Simplecrawler can handle the basic crawling quite efficiently, but like every software, it also has problems. In the future, a crawler specifically developed for web store crawling might be implemented. One of the biggest drawbacks of the Simplecrawler is its data structure for web page links. The Simplecrawler stores the whole header of the web request, even though only the URLs are needed to determine the similarity of two web pages. As a single web store can contain tens of thousands of products, the headers of every page can accumulate to memory consumption of gigabytes on a single web store. A simple optimization would be to reduce the stored data to bare minimum.

Another optimization feature would be to implement a hierarchy for the data structure holding the links. The data structure would thus mimic the web page hierarchy of the web store. This would help extracting the category information of a product. In the current version, the category of a product is interpreted from the breadcrumb on the web page. Breadcrumb is the navigation path to the page and consists of a list of navigation links. Breadcrumb usually helps the shopper to navigate back and forth between products and categories. The breadcrumb is not ubiquitous across web stores. A hierarchy in the data structure, which would tell how the page was found, could help in defining the category of a product more reliably.

The implemented product parser appears to perform very efficiently. Even the base class implementation was able to parse multiple web stores. When configured correctly the performance of the parser seems excellent and there is not any observable bottlenecks. In the future, if there seems to be a lot of web stores that can not be configured to work with the base class parser, descendent classes have to be made to handle these special cases.

7. CONCLUSION

The goal of this thesis was to investigate and implement a method to easily acquire product data from multiple online stores. At first, the goal was investigated and possible problems were sought and brought to attention. After defining problems and possible solutions, implementation was started. Used framework, programming language and different software development architectures and patterns were introduced and their use was justified. Then the implementation of a web store product scraper was done and tested.

The thesis goal was achieved. The product scraping is possible, although quite time consuming on larger web stores with tens of thousands products. The system resource requirements of the scrapers were moderate, so although a single store scrape takes a long time, multiple stores can be scraped simultaneously. The implementation is also easy to configure to support multiple different web stores. Used programming language and software development architectures seem to have been right choices. Big problems nor obstacles with the implementation were not encountered. Instead, the implementation was quite straight forward and rapid.

Tests and evaluations done for the software scraper shows that the amount of product information scraped from the stores stays stable and almost all information from products can be acquired with the first scrape. The tests also showed that the biggest slowing factor of the scraper is the web server of the targeted web store. The web server might systematically throttle network connections, just be slow or under a lot of stress. Because of this, it seems useful to spend some time optimising the configurations of each web store to not unnecessarily stress the web server of the store and to not use unnecessary system resources. Also, as consecutive scrapes do not seem to produce that much more product information, the consecutive scrapes should be limited to be performed once a day at most, to preserve resources on both ends.

The performance problems of the scraper seem to be caused by slow network infrastructure or from the network throttling of the web server on the web store side. Not that much future work is needed to optimise the performance of the scraper. Instead, the future development should be concentrated on improving the reliability of the scraper. Also, a framework should be built around the product scraper to ease the configuration of web stores and to control the scraping of product information.

The framework should also be able to monitor and log any errors and statistics of scrapes. Configuring a single web store takes time. During the implementation about 10 - 15 min average was achieved on a single store. Because of this, if the scraper is to be used widely on different web stores, some automation for configuring the stores should be implemented. As most web stores have a quite similar structure on their web pages and HTML code, some level of automation can possibly be achieved.

REFERENCES

[1] V. Sehgal, "Forrester Research Online Retail Forecast, 2013 To 2018 (US)," *Forrester Inc, ForecastView Spreadsheet*, 2015.

[2] AWS, "Reference Architectures - E-commerce Web Site." , [Online], https://media.amazonwebservices.com/architecturecenter/AWS_ac_ra_ecommerce_webfrontend_14.pdf. [Read 10.5.2015].

[3] Zalando, "Affiliate programme." , [Online], https://www.zalando.co.uk/partner/. [Read 10.5.2015].

[4] Zanox, "Homepage." , [Online], http://www.zanox.com. [Read 10.5.2015].

[5] W3C, "XML Core Working Group Public Page." , [Online], http://www.w3.org/XML/Core/. [Read 13.5.2015].

[6] W3C, "Extensible Markup Language (XML) 1.0 (Fifth Edition)." , [Online], http://www.w3.org/TR/REC-xml/. [Read 13.5.2015].

[7] Wikipedia, "HTML." , [Online], http://en.wikipedia.org/wiki/HTML. [Read 02.04.2014].

[8] W3C, "HTML5." , [Online], http://www.w3.org/TR/html/. [Read 13.5.2015].

[9] T. Bray, "The javascript object notation (json) data interchange format," 2014. [Read 12.02.2015].

[10] "JSON Homepage." , [Online], http://json.org. [Read 12.02.2015].

[11] Node.js, "Homepage." , [Online], http://nodejs.org/. [Read 16.03.2014].

[12] M. Cantelon, T. Holowaychuk, and N. Rajlich, *Node. js in Action*. Manning, 2014.

[13] D. Crockford, *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[14] Node.js, "Assert API Documentation." , [Online], https://nodejs.org/api/assert.html. [Read 16.03.2014].

[15] Should.js, "Github repository." , [Online], https://github.com/shouldjs/should.js. [Read 16.03.2014].

[16] Mocha.js, "Homepage." , [Online], http://mochajs.org/. [Read 16.03.2014].

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1995.

[18] H. He, “What is service-oriented architecture,” *Publicação eletrônica em*, vol. 30, pp. 1–5, 2003.

[19] B. M. Michelson, “Event-driven architecture overview,” *Patricia Seybold Group*, vol. 2, 2006.

[20] W. Khattak and V. Narayanan, “SOA Pattern (#11): Event-Driven Messaging.” , [Online], `http://www.informit.com/articles/article.aspx?p=1577450`, 2010. [Read 05.10.2014].

[21] E. F. Codd, “A relational model of data for large shared data banks,” *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[22] T. Haerder and A. Reuter, “Principles of transaction-oriented database recovery,” *ACM Comput. Surv.*, vol. 15, pp. 287–317, Dec. 1983.

[23] T. M. Connolly and C. E. Begg, *Database systems: a practical approach to design, implementation, and management.* Pearson Education, 2005.

[24] Wikipedia, “NoSQL.” , [Online], `http://en.wikipedia.org/wiki/NoSQL`. [Read 27.05.2014].

[25] A. Moniruzzaman and S. A. Hossain, “Nosql database: New era of databases for big data analytics-classification, characteristics and comparison,” *International Journal of Database Theory & Application*, vol. 6, no. 4, 2013.

[26] M. Stonebraker, “Sql databases v. nosql databases,” *Communications of the ACM*, vol. 53, no. 4, pp. 10–11, 2010.

[27] Github, “Simplecrawler.” , [Online], `https://github.com/cgiffard/node-simplecrawler`. [Read 06.06.2014].

[28] J. Marini, *Document Object Model.* McGraw-Hill, Inc., 2002.

[29] E. A. Meyer, *CSS: the definitive guide.* " O’Reilly Media, Inc.", 2006.

[30] MongoDB, “Homepage.” , [Online],`http://www.mongodb.org`. [Read 23.05.2014].

[31] MongooseJS, “Homepage.” , [Online], `mongoosejs.com`. [Read 15.06.2014].